

Can Formal Specifications Be Synthesized from Tests Alone?

Tianhai Liu
Karlsruhe Institute of Technology
Karlsruhe, Germany
tianhai.liu@kit.edu

Maximilian Müller
Karlsruhe Institute of Technology
Karlsruhe, Germany
maximilian.mueller7@student.kit.edu

Tobias Hey
Karlsruhe Institute of Technology
Karlsruhe, Germany
hey@kit.edu

Vitus Lüntzel
Karlsruhe Institute of Technology
Karlsruhe, Germany
luentzel@kit.edu

Muhammad Minhas
Karlsruhe Institute of Technology
Karlsruhe, Germany
minhas@kit.edu

Anne Koziolek
Karlsruhe Institute of Technology
Karlsruhe, Germany
koziolek@kit.edu

Bernhard Beckert
Karlsruhe Institute of Technology
Karlsruhe, Germany
beckert@kit.edu

Abstract

Formal specifications offer strong guarantees, but remain costly to write manually. Recent LLM-based approaches automate this by inferring specifications from source code, yet their reliance on white-box access poses barriers to industrial adoption due to intellectual property risks and deployment costs. Our approach uses LLMs to infer candidate specifications solely from test code and dynamic execution traces: the LLM observes only the program interface, selected inputs, and corresponding outputs or state changes, while the implementation internals remain hidden. Candidate specifications are validated locally using bounded model checking, with feedback guiding iterative refinement. Initial results on the SpecGenBench benchmark suggest that tests can guide LLMs towards meaningful Java Modeling Language specifications, while also highlighting checker compatibility and diagnostic feedback as key challenges for reliable refinement.

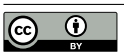
CCS Concepts

• **Software and its engineering** → **Formal software verification**; **Software testing and debugging**.

Keywords

Test-Derived Formal Specifications, LLM, BMC

ACM Reference Format:
Tianhai Liu, Maximilian Müller, Tobias Hey, Vitus Lüntzel, Muhammad Minhas, Anne Koziolek, and Bernhard Beckert. 2026. Can Formal Specifications Be Synthesized from Tests Alone?. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3832783.3834549



This work is licensed under a Creative Commons Attribution 4.0 International License.
ASE '26, Munich, Germany
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834549

1 Introduction

Unit test suites are the backbone of industrial regression testing, yet they inherently capture only *concrete* executions. As systems evolve, this granularity gap allows *silent semantic regressions* to remain undetected. While formal specifications (e.g., pre- and post-conditions) could bridge this gap by enabling broader behavioral reasoning, they are rarely available in practice due to the manual effort and expert knowledge required to construct them.

Recent efforts to automate specification synthesis have predominantly leveraged Large Language Models (LLMs) in a *white-box* setting, inferring properties directly from source code [4, 6, 9, 14, 22, 29, 30]. While effective, this approach may face practical barriers in industrial contexts. Our motivation stems from observations of long-lived industrial systems, such as those maintained by small and medium-sized enterprises (SMEs), where proprietary technology is central to their market differentiation. For such systems, maintaining test cases as the software evolves is increasingly challenging, and formal specifications could provide a more durable basis for capturing and checking behavioral expectations. However, adopting white-box LLM inference must be balanced against the need to protect confidential implementation knowledge and preserve long-term competitiveness. Moreover, source code may be unavailable to the specification-inference process altogether: security policies may restrict source code access to isolated environments, third-party components or remote services may expose only APIs, and obsolete or legacy systems may survive as binaries after build environments or original source repositories have been lost. Exposing implementations to external LLMs may be unacceptable, even under contractual safeguards, while maintaining local, state-of-the-art infrastructure is often impractical for SMEs.

These constraints lead us to the research question: *Can formal specifications be synthesized from tests and their execution observations?* We assume that such long-lived system implementations are trusted as the reference behavior. As a first step, we study whether the behavioral knowledge encoded in test code and execution traces can guide LLMs towards meaningful formal specifications without exposing implementation code to the LLM.

In our approach, the program is treated as a black box from the external LLM's perspective: the model is given the program interface, selected inputs, and observed return values or state changes, but not the implementation internals. Test code is provided to the model, enabling it to exploit oracle assertions, method-call context, and other test-level cues during specification inference. To complement this static test-side information, executions of the selected tests are encoded as structured observation traces that capture program states before and after the method call under test, including state changes that may not be explicit in the test code. The resulting candidates are then iteratively validated and refined in a counterexample-guided loop using bounded model checking (BMC), where the program is *locally* analyzed in a *white-box* manner.

This work presents an initial investigation into the feasibility of LLM-based formal specification synthesis from tests alone. Our **contributions** are: (1) An exploration of the feasibility of decoupling specification inference from source code access, utilizing only test artifacts and execution traces, (2) a proposed approach that combines oracle information from test code with behavioral observations from test execution, validating candidates via BMC, (3) a preliminary evaluation on the SpecGenBench benchmark.

2 Motivating Example

Figure 1 illustrates the setting for a simplified `deposit` method. The implementation updates `balance` and an internal `checksum`, while `auditKey` and calls to `AuditService` represent confidential security or business logic that must not be disclosed. The test checks only that the balance increases, whereas the engineer-controlled observations record selected non-sensitive state changes before and after the call, e.g., sensitive fields `auditKey` are excluded.

Using test code and observations, our approach prompts an LLM to synthesize a candidate Java Modeling Language (JML) [10] contract. In this example, the model produces the candidate contract shown in Figure 1c[1]. The candidate specification is then checked locally using BMC against the implementation via a consistency check and against the observations via a coverage check (see Section 3.3). Both checks pass, indicating that the candidate is consistent with the implementation within the analyzed bounds and admits the recorded observations. Importantly, the inferred contract can capture behaviors not explicitly asserted by the tests but visible in the observations. For instance, the tests assert only the updated `balance`, whereas the observations also expose the change to `checksum`. Consequently, if the statement updating `checksum` is removed, the inferred specification detects the violation, although the original test assertions would not.

3 Proposed Approach

Our approach infers specifications from test code and execution observations. As shown in Figure 2, tests are first executed to produce test execution observations, which make relevant pre- and post-state explicit. An LLM then uses the tests and observations to synthesize a candidate JML specification. The candidate is validated locally with BMC against the implementation and the recorded observations. If validation fails, the counterexample is fed back to the LLM for refinement. Otherwise, the candidate is returned as a specification that admits the observed test behaviors and is consistent with the implementation within the analyzed bounds.

[1]Results may vary across runs and models; this example uses ChatGPT 5.4.

3.1 Test Execution Observation

We execute selected unit tests for the method under test to obtain concrete execution traces. These traces complement test code by making observed input-output and state-transition relations explicit, which can refine specifications beyond assertions alone and provide behavioral cues when interface names are anonymized. Test selection may be guided by domain expertise, coverage goals, and operational constraints. For each selected test, engineers specify which objects and fields should be recorded immediately before and after the method call. This defines a user-controlled projection of the observable program state, which can be implemented through logging or bytecode instrumentation. In the motivating example (Figure 1), we use logging statements for ease of presentation. Tests that expose sensitive information through their code, inputs, or recorded execution data are excluded.

From each execution trace, we extract a test execution *observation* that makes the behavioral evidence explicit and reusable for prompting, observation-coverage checking, and refinement (Section 3.2–3.3). Each observation corresponds to one method invocation and is represented as $obs = \langle args,\ pre,\ post,\ outcome \rangle$. The *args* records the receiver-construction arguments and method-call arguments, including parameters of parameterized tests. The *pre* and *post* components record the selected program state immediately before and after the invocation, respectively. The *outcome* component records either normal termination, including the return value, or an exception. Objects in *pre* and *post* are represented by an identifier, a class name, and field-value mappings.

3.2 Specification Inference

We employ an LLM to synthesize a method-level JML specification from the method signature, the tests that invoke the method, and the corresponding observations. The zero-shot prompt follows a two-stage structure. First, the model infers an initial contract from the test code, using test inputs and assertions as evidence for candidate preconditions, postconditions, exceptional behavior, and assignable locations. Second, the model refines this contract using the observations, in particular repeated pre/post-state relations and observed return or exception outcomes. To reduce unsupported generalization, the prompt enforces three rules. First, clauses should generalize beyond individual tests only when supported by test assertions, repeated observations, or validation feedback. Second, unsupported domain assumptions should be reported as hypotheses rather than encoded directly as JML clauses. Third, the final output should contain non-vacuous, syntactically valid JML clauses.

3.3 Specification Validation

Given a candidate specification inferred from test code and observations, we validate it using two complementary checks. The consistency check detects candidate specifications that are contradicted by the implementation within the analyzed bounds. The observation-coverage check ensures that the inferred specification does not exclude any observed test behavior.

(a) Implementation

```
class Account {
 int balance, checksum;
 String auditKey; // sensitive
 Account(int balance) {
  this.balance = balance;
  this.checksum = 7;
  this.auditKey = AuditSer.issueKey();
 }
 void deposit(int amount) {
  AuditSer.authorize(auditKey, amount);
  balance += amount;
  checksum += 31 * amount;
}}
```

(b) Simplified test code and observations

```
@ParameterizedTest
@CsvSource({"10,5", "20,3", "0,7"})
void testDeposit(int initial, int amount) {
 Account a = new Account(initial);
 // log pre: initial,amount,a.balance,a.checksum
 a.deposit(amount);
 // log post: a.balance, a.checksum
 assertEquals(initial + amount, a.balance);
}
```

```
(10,5): (10,7) -> (15,162)
(20,3): (20,7) -> (23,100)
(0,7):  (0,7)  -> (7,224)
```

(c) Refined candidate specification

```
/*@ public normal_behavior
  @ requires
  @     balance <= Integer.MAX_VALUE - amount
  @  && amount <= Integer.MAX_VALUE/31
  @  && checksum <= Integer.MAX_VALUE - 31*amount;
  @
  @ ensures balance == \old(balance) + amount;
  @ ensures checksum == \old(checksum)+ 31*amount;
  @
  @ assignable balance, checksum;
  @*/
void deposit(int amount);
```

Figure 1: Motivating example: inferring a JML specification from tests.

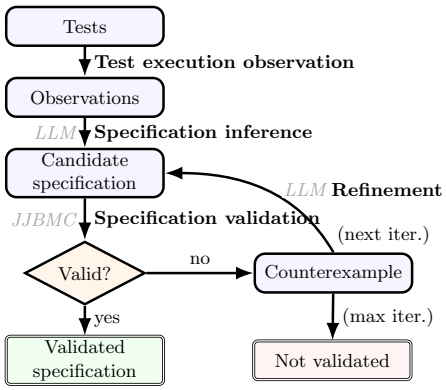


Figure 2: Overview of the specification inference pipeline.

We consider three behavior sets: (i) B_{code}, behaviors exhibited by the implementation, (ii) B_{test}, the subset exercised by the selected unit tests, and (iii) B_{spec}, behaviors admitted by the candidate specification, with $B_{test} \subseteq B_{code}$ by construction. A behavior b corresponds to a pre- and post-state pair of a method execution.

Consistency check. We treat the implementation as the local behavioral reference, reflecting our target setting of mature industrial software that has evolved over the years and is already supported by regression tests. A candidate specification should therefore not exclude behaviors that the implementation can exhibit within the bounds of analysis. We use BMC to check, within a given bound, whether the precondition and program semantics admit an execution that violates the postcondition. If such an execution exists, BMC reports it as a counterexample $b \in B_{code} \setminus B_{spec}$, showing that the specification excludes an implementation behavior and is therefore inconsistent. We convert the counterexample into an additional observation and feed it back to the LLM for refinement.

Observation-coverage check. The consistency check ensures that the candidate specification admits all implementation behaviors within the analyzed bound, e.g., bounded loop unrollings. If this bound is sufficient, the observed execution behaviors would already be admitted by the specification. However, tests may exercise behaviors outside the BMC search space. We therefore perform an observation-coverage check that directly checks whether each recorded test behavior is admitted by the candidate specification.

We encode observations as JML-annotated harness specifications that fix observed pre-states and require the corresponding post-states. The target method is interpreted contractually rather than inlined, making the check independent of the BMC loop-unrolling bound. A counterexample $b \in B_{test} \setminus B_{spec}$ means that the contract excludes an observed behavior and guides refinement. For example, an observation $balance : 10 \mapsto 15$ under $amount = 5$ means that the contract allows a call starting from $a.balance = 10$ with $amount = 5$ and ending in $a.balance = 15$. A candidate with `requires amount > 10` rejects this call and therefore fails the coverage check.

A candidate that passes both checks is treated as a validated candidate, not as a complete specification. The checks rule out two concrete failure modes: contradicting the implementation within the analyzed bounds and excluding recorded test behaviors. They do not establish that all unobserved behaviors are specified precisely; specification strength is therefore assessed separately.

Table 1: Preliminary results on the 22 SpecGenBench tasks.

Task	History	Result	Task	History	Result
abs	11	Success	binarySearch	00	Error
biggest	11	Success	bubbleSortDesc	00	Error
calculate	11	Success	arrcmp	00	Error
canWinNim	11	Success	containsDuplicate	00	Error
conjunctOf	11	Success	convertToFahrenheit	00	Error
disjunctOf	11	Success	convertToKelvin	00	Error
divisorGame	11	Success	convertToTitle	00	Error
addLoop	01,11	Success	copyArray	00	Error
checkString	01,11	Success	digitRoot	00	Error
computeOverlapArea	01,11	Success	bubbleSort	00,00,00	Fail
convertTemperature	10,10,11	Success	changeCase	10,10,10	Fail

4 Preliminary Results

We conducted an initial feasibility study on SpecGenBench [14]. SpecGenBench was originally designed for specification inference from source code, where multiple implementations of the same API provide complementary evidence about the intended behavior. In contrast, our approach infers specifications from tests and execution observations without exposing the implementation source code. We therefore evaluate our approach at the API level, treating each API as a separate target-specification task.

For each API, we use *gpt-oss-120b* [18] to generate JUnit tests. Across the evaluated tasks, these tests achieve an average instruction coverage of 76.9%, providing a coarse sanity check that the inferred specifications are not based solely on trivial executions. We use GPT-5.4 [17] to infer and refine candidate JML specifications. Each candidate is checked with JJBMC [2] using its default loop-unrolling bound of 5 against two criteria: implementation consistency and observation admission. Such bounds are a known source of precision concerns in bounded verification [13]. Failed

candidates are refined using validation feedback for up to 3 iterations per task. We evaluate our prototype on 22 SpecGenBench APIs compatible with JJBMC. Their implementations use Booleans, integers, floating-point values, arrays, and loops. We exclude tasks that require features JJBMC doesn't support, such as strings or reachability-style heap properties.

Table 1 shows the evaluation results. We classify the outcomes as follows: Success means that the final candidate passes both the consistency and observation-coverage checks; Error means that the generated JML is not supported by JJBMC; and Fail means that the refinement iteration bound is reached without obtaining a candidate that passes both checks. In the history column, each iteration is encoded as a two-bit outcome: the first bit denotes the consistency check, and the second denotes the observation-coverage check; 1 indicates success and 0 indicates failure. For example, 01, 11 means that consistency fails but coverage passes in the first iteration, while both checks pass in the second.

The prototype succeeds on 11 of the 22 tasks. Among these, 7 are solved in the first iteration, while 4 require refinement. The `convertTemperature` case is representative of the refinement. The initial specification expressed the expected conversion results as postcondition `result[0] = celsius + 273.15`, where `celsius` is the parameter of the tested method. During refinement, JJBMC reported a consistency violation caused by unsafe JML expressions, such as dereferencing `\result` or accessing array elements without guards. Using this feedback, the LLM added well-definedness guards `\result != null && \result.length >= 1 ⇒ \result[0] == celsius + 273.15`. Such guards avoid null dereferences and out-of-bounds accesses during JML evaluation.

The dominant limitation is compatibility with the JML fragment supported by JJBMC. Among the 22 tasks, 9 terminate with an *Error* because the generated specifications contain constructs outside the fragment supported by JJBMC, such as `\num_of`, ghost or model fields, and arithmetic expressions over quantified variables. Since these constructs are valid in standard JML but cannot be validated by the current toolchain, we treat these cases primarily as checker-compatibility failures rather than semantic failures of the inferred specifications after closer manual investigation.

Checker interaction also explains the two *fail* cases. These tasks reach the refinement bound without obtaining a candidate that is both consistent with the implementation and covers the observations. In these cases, JJBMC reports failed checks but does not provide an informative counterexample, returning only an assertion failure with an empty concrete example. Consequently, the LLM receives little actionable feedback and can only guess how to refine the candidate specification.

The results provide early evidence for the feasibility of inferring JML specifications from test code and observations, while identifying checker-aware specification generation as the main current challenge. Future work will therefore provide checker-aware pre-checking and a repair layer, and improve JJBMC.

A threat to validity is the potential leakage of benchmark data. SpecGenBench tasks or similar code fragments may have appeared in LLM pre-training data. We plan to mitigate this risk by evaluating the approach on proprietary industrial software whose implementations and intended specifications are not publicly available, and by anonymizing the method and test code in SpecGenBench.

5 Related Work

Dynamic specification inference derives likely properties from observed executions. Daikon [5] pioneered this line by mining invariants from instrumented runs, while later work on specification mining inferred temporal rules, API protocols, behavioral patterns, and algebraic specifications from execution traces and program interactions [1, 8]. Subsequent approaches extended the predicate space with quantified properties, implication-style specifications, or evolutionary search [15, 16, 27]. These techniques are related to our use of observations, but they typically rely on predefined templates, handcrafted predicates, or trace mining rather than exploiting test-code oracle structure and LLM-based generalization.

Specification synthesis approaches such as AutoSpec [29], SpecGen [14], and Allo2JML [7], as well as works on loop invariant generation [3, 4, 21, 25] and class invariant generation [24], demonstrate the potential of automated specification synthesis.

Granberry et al. [6] further enrich prompts with generated tests and static analysis results. In contrast, our work explores a complementary setting in which the LLM receives controlled test code and sanitized observations, while implementation code is excluded from the prompt and used only for local validation.

Test- and oracle-based techniques infer executable assertions or postconditions from generated executions or transformation traces [31]. Like our approach, they treat executions as evidence of intended behavior. However, they usually aim to strengthen test oracles or learn assertions, rather than to synthesize JML method contracts. Our approach uses test code structure and heap observations as evidence for LLM-based generalization, followed by local JML validation.

6 Conclusion and Future Work

We investigated whether LLMs can synthesize formal specifications from tests alone. Our framework extracts structured execution observations, combines them with test-oracle information to generate candidate JML specifications, and validates them through BMC-based counterexample-guided refinement. This behavior-oriented approach avoids reliance on source-code structure. On 22 SpecGen-Bench API-level tasks, our prototype automatically generated and validated specifications for 11 using JJBMC.

Test-level evidence provides partial behavioral coverage. To offset missing source-code context, we plan to incorporate public documentation [32], API specifications [19], and usage examples [23] to capture intent and resolve behavior under-specified by tests.

We will also address scalability by summarising large traces into representative behavioral patterns before inference. To mitigate BMC state-space explosion, we plan to combine bounded validation with program slicing [26, 28] and incremental symbolic execution [11, 20].

Acknowledgements

This work was supported by the German Research Foundation (DFG) – SFB 1608 – 501798263 and by the pilot program Core Informatics at KIT (KiKIT) of the Helmholtz Association (HGF).

Data Availability Statement

All benchmark, evaluation data, and prototype artifacts are publicly available at [12].

References

[1] Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining Specifications. *ACM SIGPLAN Notices* 37, 1 (2002), 4–16. doi:10.1145/565816.503275

[2] Bernhard Beckert, Michael Kirsten, Jonas Klamroth, and Mattias Ulbrich. 2020. Modular Verification of JML Contracts Using Bounded Model Checking. In *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, Vol. 12476. Springer, 60–80. doi:10.1007/978-3-030-61362-4_4

[3] Bernhard Beckert, Jonas Klamroth, Wolfram Pfeifer, Patrick Röper, and Samuel Teuber. 2025. Towards Combining the Cognitive Abilities of Large Language Models with the Rigor of Deductive Program Verification. In *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 242–257. doi:10.1007/978-3-031-75387-9_15

[4] Saikat Chakraborty, Shuvendu Lahiri, Sarah Fakhoury, Akash Lal, Madanlal Musuvathi, Aseem Rastogi, Aditya Senthilnathan, Rahul Sharma, and Nikhil Swamy. 2023. Ranking LLM-Generated Loop Invariants for Program Verification. In *Findings of the Association for Computational Linguistics: The 2023 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 9164–9175. doi:10.18653/v1/2023.findings-emnlp.614

[5] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 1999. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *International Conference on Software Engineering*. ACM, 213–224. doi:10.1145/302405.302467

[6] George Granberry, Wolfgang Ahrendt, and Moa Johansson. 2025. Specify What? Enhancing Neural Specification Synthesis by Symbolic Methods. In *Integrated Formal Methods*. Springer, 307–325. doi:10.1007/978-3-031-76554-4_19

[7] Daniel Grunwald, Christoph Gladisch, Tianhai Liu, Mana Taghdiri, and Shmuel S. Tyszberowicz. 2014. Generating JML Specifications from Alloy Expressions. In *Haifa Verification Conference*. Springer, 99–115. doi:10.1007/978-3-319-13338-6_9

[8] Johannes Henkel and Amer Diwan. 2003. Discovering Algebraic Specifications from Java Classes. In *European Conference on Object-Oriented Programming*. Springer, 431–456. doi:10.1007/978-3-540-45070-2_19

[9] Thanh Le-Cong, Bach Le, and Toby Murray. 2025. Can LLMs Reason About Program Semantics? A Comprehensive Evaluation of LLMs on Formal Specification Inference. In *Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 21991–22014. doi:10.18653/v1/2025.acl-long.1068

[10] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. 2006. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes* 31, 3 (May 2006), 1–38. doi:10.1145/1127878.1127884

[11] Tianhai Liu, Mateus Araújo, Marcelo d'Amorim, and Mana Taghdiri. 2014. A Comparative Study of Incremental Constraint Solving Approaches in Symbolic Execution. In *Haifa Verification Conference*, Vol. 8855. Springer, 284–299. doi:10.1007/978-3-319-13338-6_21

[12] Tianhai Liu, Maximilian Müller, Tobias Hey, Muhammad Minhas, Vitus Lüntzel, Anne Koziolek, and Bernhard Beckert. 2026. Dataset and Artifacts for Can Formal Specifications Be Synthesized from Tests Alone? doi:10.5281/zenodo.21249653

[13] Tianhai Liu, Shmuel S. Tyszberowicz, Bernhard Beckert, and Mana Taghdiri. 2017. Computing Exact Loop Bounds for Bounded Program Verification. In *International Symposium on Theoretical Aspects of Software Engineering*. Springer, 147–163. doi:10.1007/978-3-319-69483-2_9

[14] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2025. SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. In *International Conference on Software Engineering*. IEEE, 16–28. doi:10.1109/ICSE55347.2025.00129

[15] Bertrand Meyer, Ilinca Ciupa, Andreas Leitner, and Lisa Ling Liu. 2007. Automatic Testing of Object-Oriented Software. In *International Conference on Current Trends in Theory and Practice of Computer Science*. Springer, 114–129. doi:10.1007/978-3-540-69507-3_9

[16] Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo Frias. 2021. EvoSpex: An Evolutionary Algorithm for Learning Postconditions. In *International Conference on Software Engineering*. IEEE, 1223–1235. doi:10.1109/ICSE43902.2021.00112

[17] OpenAI. 2026. GPT-5.4 Model. https://developers.openai.com/api/docs/models/gpt-5.4. Accessed: 2026-05-12.

[18] OpenAI. 2026. gpt-oss-120b Model. https://developers.openai.com/api/docs/models/gpt-oss-120b. Accessed: 2026-05-12.

[19] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. 2012. Inferring Method Specifications from Natural Language API Descriptions. In *International Conference on Software Engineering*. IEEE, 815–825. doi:10.1109/ICSE.2012.6227137

[20] Corina S. Păsăreanu and Willem Visser. 2009. A Survey of New Trends in Symbolic Execution for Software Testing and Analysis. *International Journal on Software Tools for Technology Transfer* 11, 4 (2009), 339–353. doi:10.1007/s10009-009-0118-1

[21] Muhammad A. A. Pirzada, Giles Reger, Ahmed Bhayat, and Lucas C. Cordeiro. 2024. LLM-Generated Invariants for Bounded Model Checking Without Loop Unrolling. In *International Conference on Automated Software Engineering*. 1395–1407. doi:10.1145/3691620.3695512

[22] Armand Puccetti, Gaël de Chalendar, and Pierre-Yves Gibello. 2021. Combining Formal and Machine Learning Techniques for the Generation of JML Specifications. In *Workshop on Formal Techniques for Java-like Programs*. ACM, 59–64. doi:10.1145/3464971.3468425

[23] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. *ACM SIGPLAN Notices* 49, 6 (2014), 419–428. doi:10.1145/2666356.2594321

[24] Chuyue Sun, Viraj Agashe, Saikat Chakraborty, Jubi Taneja, Clark Barrett, David Dill, Xiaokang Qiu, and Shuvendu K. Lahiri. 2026. ClassInvGen: Class Invariant Synthesis Using Large Language Models. In *AI Verification*. Springer, 64–96. doi:10.1007/978-3-031-99991-8_4

[25] Samuel Teuber and Bernhard Beckert. 2025. Next Steps in LLM-Supported Java Verification. In *International Workshop on Next-Generation Software Engineering*. IEEE, 1–4. doi:10.1109/NSE66660.2025.00007

[26] Frank Tip. 1995. A Survey of Program Slicing Techniques. *Journal of Programming Languages* 3, 3 (1995), 121–189.

[27] Yi Wei, Carlo A. Furia, Nikolay Kazmin, and Bertrand Meyer. 2011. Inferring Better Contracts. In *International Conference on Software Engineering*. ACM, 191–200. doi:10.1145/1985793.1985820

[28] Mark Weiser. 1984. Program Slicing. *IEEE Transactions on Software Engineering* SE-10, 4 (1984), 352–357.

[29] Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. 2024. Enchanting Program Specification Synthesis by Large Language Models Using Static Analysis and Program Verification. In *International Conference on Computer Aided Verification*. Springer, 302–328. doi:10.1007/978-3-031-65630-9_16

[30] Guangyuan Wu, Weining Cao, Yuan Yao, Hengfeng Wei, Taolue Chen, and Xiaoxing Ma. 2024. LLM Meets Bounded Model Checking: Neuro-symbolic Loop Invariant Inference. In *International Conference on Automated Software Engineering*. ACM, 406–417. doi:10.1145/3691620.3695014

[31] Tao Xie. 2006. Augmenting Automatically Generated Unit-Test Suites with Regression Oracle Checking. In *ECOOP 2006 – Object-Oriented Programming*. Springer, 380–403. doi:10.1007/11785477_23

[32] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. 2011. Inferring Specifications for Resources from Natural Language API Documentation. *Automated Software Engineering* 18 (2011), 227–261. doi:10.1007/s10515-011-0082-3

Received 2026-05-12; accepted 2026-07-02